2INC0 - Operating Systems

Atomicity and interference

Geoffrey Nelissen

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e
Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Announcements

- First homework: deadline on Sunday
- New homework released every weekend

- Topics covered by the final exam:
  - preparatory material
  - lecture content
  - additional exercises

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2 and 3)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - action synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - deadlock (lecture 7)
- **File systems** (lecture 8)
- **Memory management** (lectures 9 and 10)
- **Input/output** (lecture 11)

Agenda

- Reminder of lectures 2-3
- Atomicity, Traces and concurrency
- Synchronization
- Peterson’s algorithm
- Synchronization with mutexes

Processes

- Let *x* be a ***global*** variable initialized to 0.
- Let Task1 and Task2 be POSIX processes executing the same program concurrently.
- Task2 is the child of Task1.
- What is the final value of *x* in Task1 and Task2?

Task1: |[*x := x+1;*]| || *Task2:* |[*x := x+3;*]|

Answer:
- 1 in Task1
- 3 in Task2

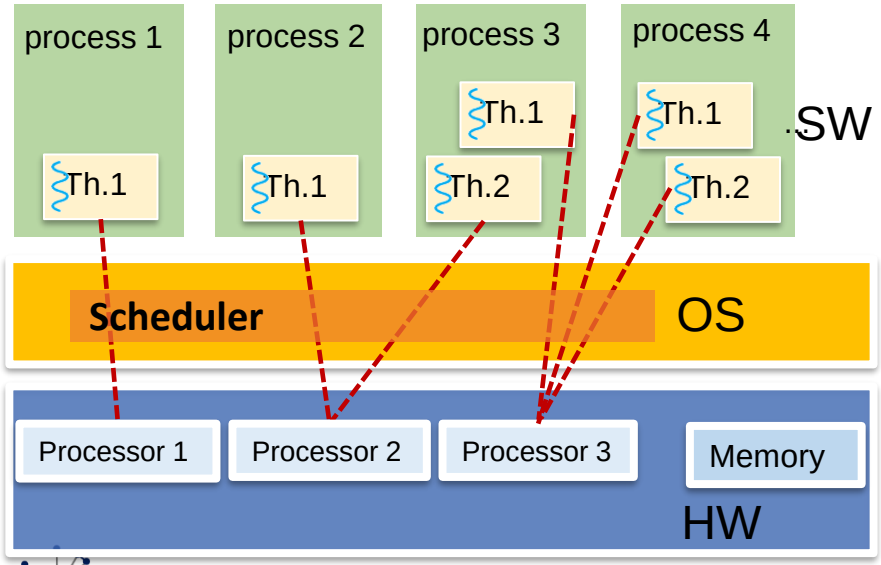
Ingredients of concurrent execution: processes and threads

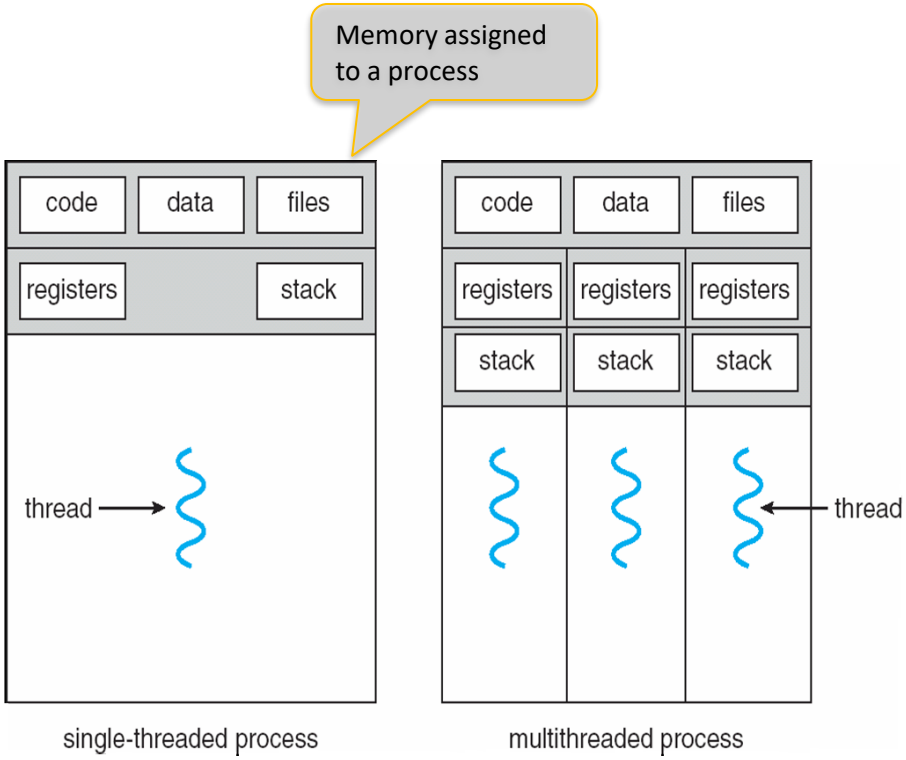
Process

- It is a **program in execution**.
- It has a context of execution
- A process owns resources (has memory and can own other resources such as files, etc.)

Thread

- A **dispatchable unit of work within a process**
- Threads within a process share code and data segments (i.e., share memory address space)





Threads

- Let *x* be a ***global*** variable initialized to 0.
- Let Task1 and Task2 be threads executing the same program concurrently.
- What is the final value of *x* in Task1 and Task2?

Task1: |[*x := x+1;*]| || *Task2:* |[*x := x+3;*]|

Answer:
x = 1 or 3 or 4
for both Task1
and Task2

Agenda

- Reminder of lecture 2+3
- Atomicity, Traces and concurrency
- Synchronization

Atomicity and traces

Atomic action

Trace

Concurrent trace

- **Indivisible** step in the evolution of an executed program
  (outcome only depends on system state when the action starts and not on what runs concurrently during the execution of the action)

How can we know if a statement is atomic?

Atomicity and traces

TU/e

Atomic action

Trace

Concurrent trace

- **Indivisible** step in the evolution of an executed program
  (outcome only depends on system state when the action starts and not on what runs concurrently during the execution of the action)

Single reference rule:

a statement (expression) may be regarded as **atomic** **if it makes** **at most one reference to a shared variable**.

IRiS

Atomicity

- If *x* and *y* are **shared** variables and *z* is a **private** variable, which of the following statements can be considered as atomic

1. x := x+y; 
2. x := y*z; 
3. z := z+1; 
4. z := z+z*y; 
5. y++; 

***Note**: if we write <x:=x+z>*
(placing the statement between < ... >)
then the statement should be considered to be atomic

Atomicity and traces

TU/e

Atomic action

- **Indivisible** step in the evolution of an executed program

Trace

- **Sequence of atomic actions** in the execution of a program
- Typically, a sequence of **assignments** and **tests**

Concurrent trace

- Trace made of an **interleaving** of atomic actions of two or more tasks

Single reference rule:

a statement (expression) may be regarded as **atomic** **if it makes** **at most one reference to a shared variable**.

IRiS

Threads: proving our answer with traces

- Let *x* be a ***global*** variable **initialized to 0.**
- **Let Task1 and Task2 be threads executing concurrently.**
- **What is the final value of *x*?**

Task1: |[*x := x+1;*]| || *Task2:* |[*x := x+1;*]|

We use two additional variables, ***reg1*** and ***reg2*** to represent the **content of the internal processor registers** used by *Task1* and *Task2*, respectively.

Possible trace 1: (reg2:=x)(reg2 := reg2+1)(x := reg2) {x=1} (reg1:=x) {reg1=1} (reg1 := reg1+1)(x := reg1) {x=2}

Possible trace 2: (reg1:=x) {reg1=0} (reg2:=x) {reg2=0} (reg2 := reg2+1) {reg2=1} (x := reg2) {x=1}
(reg1 := reg1+1) {reg1=1} (x := reg1) {x=1}

…

Note: for a **complete proof**, we would need to **write all possible concurrent traces** and show they all lead to x=1 or x=2

Answer:
x = 1 or 2

Ready for an adventure?

1. Form a group with the people around you.

2. Together, solve the problem that will be shown on the next slide.
   - You have 10 minutes.

Anime: One piece
Character: Luffy

Exercise on concurrency

```
#include <stdio.h>
#include <pthread.h>

int y;

void Count_10 ()
{
  for (int i = 0; i < 10; i ++) {
    y = y+1;
  }
}

void main()
{
  pthread_t thread_id;

  y = 0;
  pthread_create (&thread_id, NULL, Count_10, NULL);
  Count_10 ();
  pthread_join (thread_id, NULL);
}
```

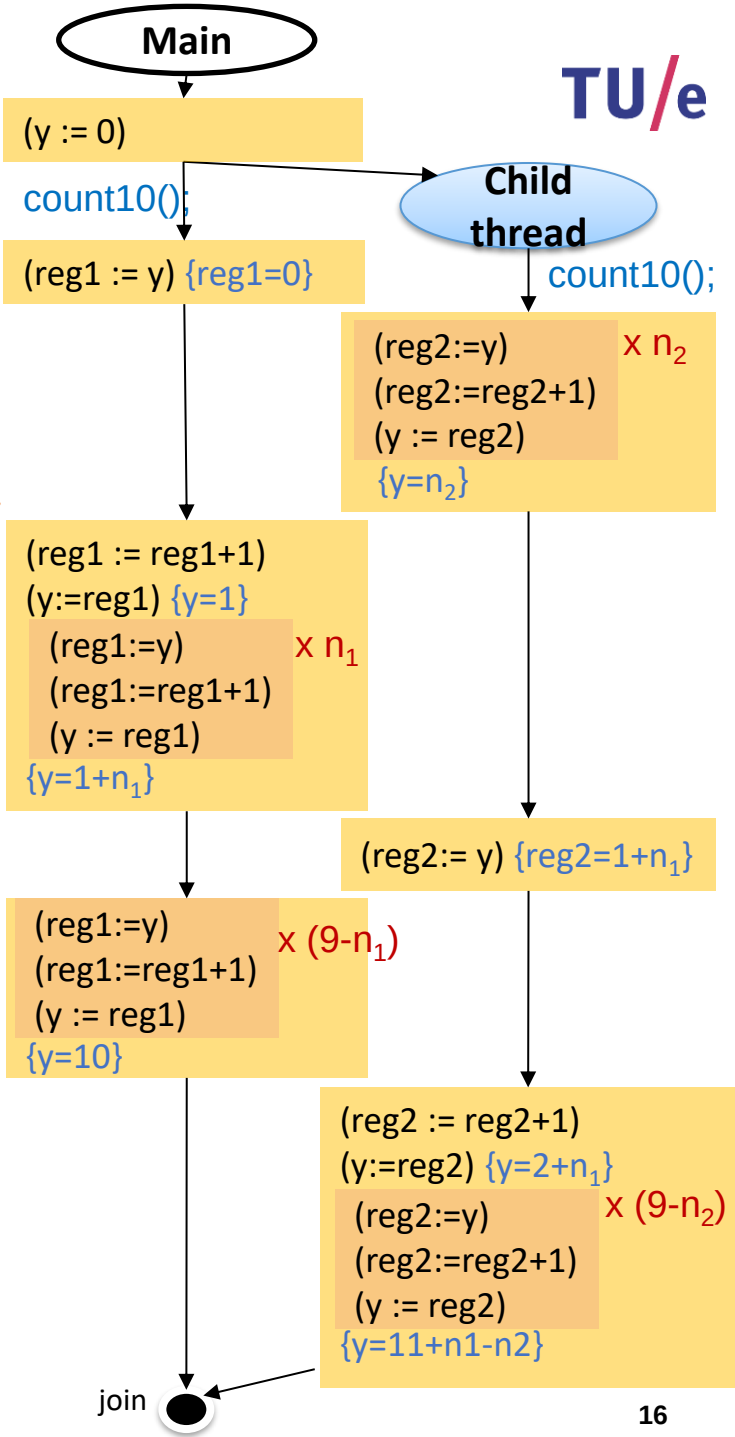
- The statement '*y=y+1' is not* atomic.
- What are the possible final values of *y?*
- ***Use traces*** *to prove your answer*

Exercise on concurrency

```
#include <stdio.h>
#include <pthread.h>

int y;

void Count_10 ()
{
  for (int i = 0; i < 10; i ++) {
    y = y+1;
  }
}

void main()
{
  pthread_t thread_id;

  y = 0;
  pthread_create (&thread_id, NULL, Count_10, NULL);
  Count_10 ();
  pthread_join (thread_id, NULL);
}
```

In this trace, n1 and n2 can take any value between 0 and 9

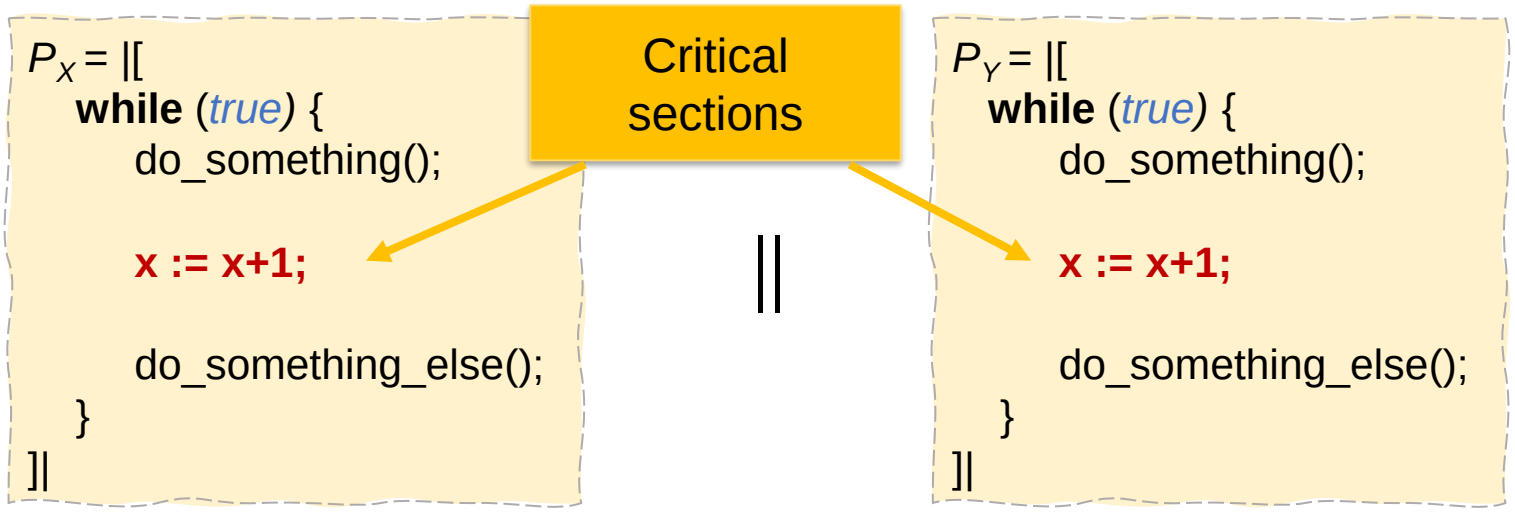
Answer: y is equal to any number between 2 and 20



Agenda

- Reminder of lectures 2+3
- Atomicity, Traces and concurrency
- Synchronization
- Peterson's algorithm
- Synchronization with mutexes

Synchronize two tasks



Our synchronization goal:
How can we coordinate the execution of the given programs such that no two tasks execute the statement *x:=x+1* at then same time?

Mutual exclusion

Synchronization refers to *ordering the execution*, i.e., forbidding certain traces
(more complex synchronization goals addressed in the next two lectures)

Synchronize through booleans

- Introduce global boolean variables *bX* and *bY* to record when we are in the critical section
  - Initially: *bX=false and bY=false*

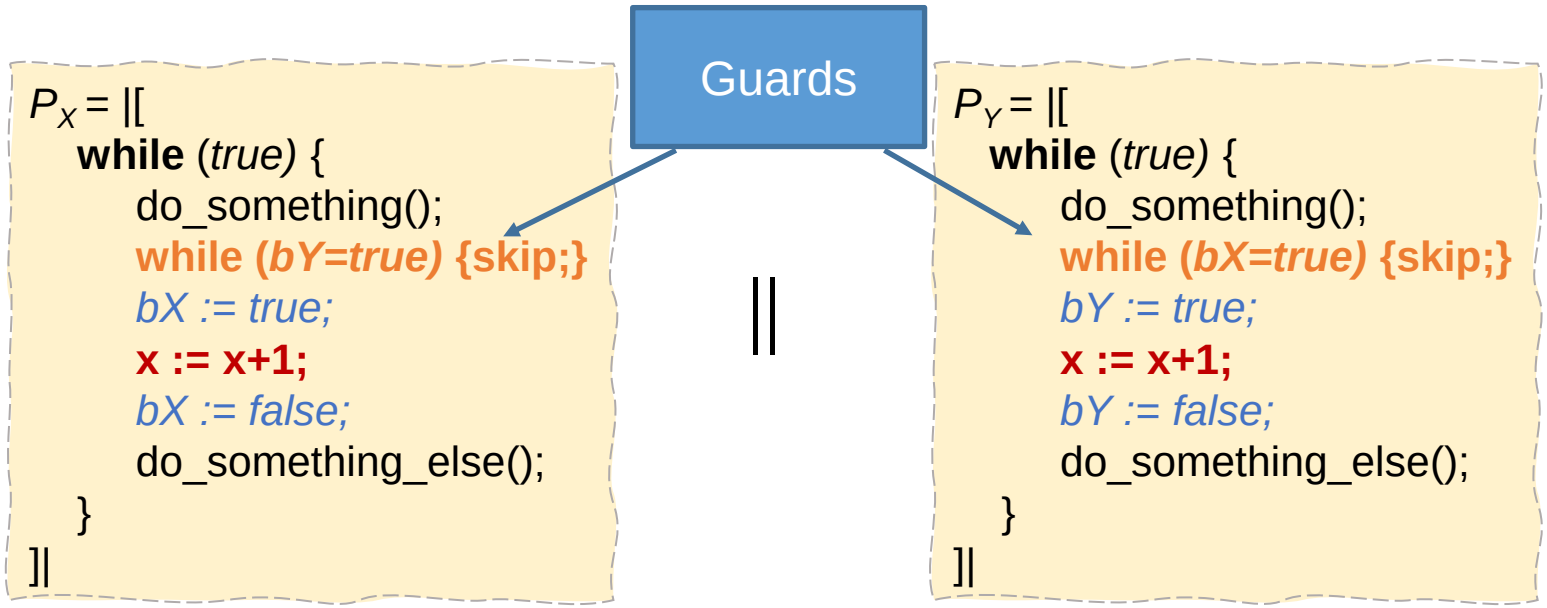
```
PX = |[
    while (true) {
        do_something();
        bX := true;
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
PY = |[
    while (true) {
        do_something();
        bY := true;
        x := x+1;
        bY := false;
        do_something_else();
    }
]|
```

Synchronize through booleans

- Introduce global boolean variables *bX* and *bY* to record when we are in the critical section
  - Initially: *bX=false and bY=false*



Does this work? Why?

Synchronize through booleans

- Introduce global boolean variables *bX* and *bY* to record when we are in the critical section

  - Initially: *bX=false and bY*

Assertions that must hold at that point of the execution for the solution to work

```
P_X = |[
    while (true) {
        do_something();
        while (bY=true) {skip;}
        bX := true;
        {bX ∧ !bY }
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
= |[
hile (true) {
        do_something();
        while (bX=true) {skip;}
        bY := true;
        {bY ∧ !bX }
        x := x+1;
        bY := false;
        do_something_else();
    }
]|
```

Assertions in both tasks falsified

Trace:{!bY∧ !bX} (!(bY=true)) (!(bX=true)) (bY:=true) (bX:=true) {bY∧ bX}

Let’s try something else…

Swap the order of the Boolean assignment and guard…

- Instead of recording when we are in the critical section, *bX* and *bY* could **record the *interest*** in entering the critical section

  - Initially: *bX=false and b*

Swap guard and assignment

```
PX = |[
    while (true) {
        do_something();
        bX := true;
        while (bY=true) {skip;}
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
PY = |[
    while (true) {
        do_something();
        bY := true;
        while (bX=true) {skip;}
        x := x+1;
        bY := false;
        do_something_else();
    }
]|
```

Does this work? Why?

Change the order…

- Instead of recording when we are in the critical section, *bX* and *bY* could **record the *interest*** in entering the critical section
  - Initially: *bX=false and bY=false*

```
PX = |[
   while (true) {
       do_something();
       bX := true;
       while (bY=true) {skip;}
       x := x+1;
       bX := false;
       do_something_else();
   }
]|
```

||

```
PY = |[
   while (true) {
       do_something();
       bY := true;
       while (bX=true) {skip;}
       x := x+1;
       bY := false;
       do_something_else();
   }
]|
```

Trace: (bX:=true)(bY:=true) {bY∧ bX}

No task can proceed: **Deadlock**

Let’s try something else…

Take turns...

- Rather than trying to **obtain** access, the task **give** the access away using a shared variable *t* that tells who can go next
  - Initially: *t = X or t = Y*

```
P_X = |[
    while (true) {
        do_something();
        t := Y;
        while( t != X){ skip;}
        x := x+1;
        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        t := X;
        while( t != Y){ skip;}
        x := x+1;
        do_something_else();
    }
]|
```

Does this work? Why?

Take turns...

- Rather than trying to **obtain** access, the task **give** the access away using a shared variable *t* that tells who can go next
  - Initially: *t = X or t = Y*

```
P_X = |[
    while (true) {
        do_something();
        t := Y;
        while( t != X){ skip;}
        x := x+1;
        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        t := X;
        while( t != Y){ skip;}
        x := x+1;
        do_something_else();
    }
]|
```

- **Functionally correct: mutual exclusion** is respected
- **No deadlock**
- **Waiting is not minimal**: a task must wait for the other to be able to enter the critical section

Requirements on synchronization solutions

- **Functional correctness:**
  - satisfy the given specification (e.g., mutual exclusion) -- an assertion that is needed for correctness must not be disturbed (by another process).
- **Minimal waiting: (“make progress”)**
  - waiting only when correctness is in danger.
- **Absence of deadlock:**
  - don’t manoeuvre (part of) the system into a state such that progress is no longer possible.
- **Absence of livelock:**
  - ensure convergence towards a decision in a synchronization protocol
- **Fairness in competition:**
  - **(weak) eventually**, each contender should be admitted to proceed.
  - **(strong)** we can put a **bound on the waiting time** of a contender.
  - absence of fairness: leads to *starvation* of tasks.

- **Note:** the last four are only required when there is concurrency

Agenda

- Reminder of lectures 2+3
- Atomicity, Traces and concurrency
- Synchronization
- **Peterson's algorithm**
- **Synchronization with mutexes**

Peterson's algorithm

Combine the ideas of the last two solutions

```
P_X = |[
    while (true) {
        do_something();

        x := x+1;

        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();

        x := x+1;

        do_something_else();
    }
]|
```

Peterson's algorithm

Combine the ideas of the last two solutions

- **take turns** in crowded circumstances (use a variable *t*)

```
P_X = |[
    while (true) {
        do_something();
        t := Y;
        while( t!=X ){ skip;}
        x := x+1;

        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        t := X;
        while( t!=Y ){ skip;}
        x := x+1;

        do_something_else();
    }
]|
```

Peterson's algorithm

Combine the ideas of the last two solutions

- **take turns** in crowded circumstances (use a variable *t*)
- **don't wait if there is no need** (look at the boolean *bY* or *bX*)

```
P_X = |[
    while (true) {
        do_something();
        bX := true;
        t := Y;
        while( bY=true ∧ t!=X ){ skip;}
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        bY := true;
        t := X;
        while( bX=true ∧ t!=Y ){ skip;}
        x := x+1;
        bY := false;
        do_something_else();
    }
]|
```

Peterson's algorithm – check correctness

We want to show:

- **mutual exclusion**
- **absence of deadlock**

We can prove it

- using traces (tedious), or
- using **detailed annotation** of the program

Check correctness – absence of deadlock

- Proof by **contradiction:**
  - ***Assume there is a deadlock***, then…

    both tasks must be blocked **on the while loop**.

  ➔ The guards must hold for both tasks at the same time

```
P_X = |[
    while (true) {
        do_something();
        bX := true;
        t := Y;
        while( bY=true ∧ t!=X ){ skip;}
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        bY := true;
        t := X;
        while( bX=true ∧ t!=Y ){ skip;}
        x := x+1;
        bY := false;
        do_something_else();
    }
]|
```

Check correctness – absence of deadlock

- Proof by **contradiction**:
  - ***Assume there is a deadlock***, then…
    both tasks must be blocked **on the while loop**.

➔ The guards must hold for both tasks at the same time

➔ For P_X: $bY \wedge t \neq X$
and for P_Y: $bX \wedge t \neq Y$

➔ $bY \wedge t \neq X \ \wedge bX \wedge t \neq Y$

= ***{$t \neq X$ and $t \neq Y$ cannot be true at the same time}***

false

➔ *We reached a contradiction*

➔ ***There is no deadlock***

Check correctness – mutual exclusion

- Proof by **contradiction:**
  - ***Assume there is no mutual exclusion***, then…
    both tasks must be in their critical section **at the same time**.

➔ **States of P_X and P_Y after their guards** must hold at the same time.

What are those states?

```
P_X = |[
    while (true) {
        do_something();
        bX := true;
        t := Y;
        while( bY=true ∧ t!=X ){ skip;}
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        bY := true;
        t := X;
        while( bX=true ∧ t!=Y ){ skip;}
        x := x+1;
        bY := false;
        do_something_else();
    }
]|
```

Check correctness – mutual exclusion

- Proof by **contradiction:**
  - ***Assume there is no mutual exclusion***, then…
    both tasks must be in their critical section **at the same time**.

➔ **States of P_X and P_Y after their guards** must hold at the same time.

What are those states?

```
P_X = |[
    while (true) {
        do_something();
        bX := true;
        (*)
        t := Y;
        while( bY=true ∧ t!=X ){ skip;}
        { bX ∧  (t = X ∨ !bY ∨ PY at (**) )}
        x := x+1;
        bX := false;
        do_something_else();
    }
]|
```

||

```
P_Y = |[
    while (true) {
        do_something();
        bY := true;
        (**)
        t := X;
        while( bX=true ∧ t!=Y ){ skip;}
        { bY ∧  (t = Y ∨ !bX ∨ PX at (*) )}
        x := x+1;
        bY := false;
        do_something_else();
    }
```

Proving that those assertions are correct is out of the scope of the course

Check correctness – mutual exclusion

- Proof by **contradiction:**
  - ***Assume there is no mutual exclusion***, then…
    both tasks must be in their critical section **at the same time**.

➔ **States of P_X and P_Y after their guards** must hold at the same time.

What are those states?

```
PX = |[
   while (true) {
        do_something();
        <bX, auxX := true, true>;
        <t, auxX := Y, false>;
        while( bY=true ∧ t!=X ){ skip;}
        { bX  ∧ (t = X ∨ !bY ∨ auxY )}
        x := x+1;
        bX := false;
        do_something_else();
   }
]|
```

||

```
PY = |[
   while (true) {
        do_something();
        <bY, auxY := true, true>;
        <t, auxY := X, false>;
        while( bX=true ∧ t!=Y ){ skip;}
        { bY  ∧  (t = Y ∨ !bX ∨ auxX )}
        x := x+1;
        bY := false;
        do_something_else();
   }
]|
```

Check correctness – mutual exclusion

- Proof by **contradiction**:
  - ***Assume there is no mutual exclusion***, then…

    both tasks must be in their critical section **at the same time**.

➔ **States of P_X and P_Y after their guards** must hold at the same time.

What are those states?

For P_X: $\{ bX \ \wedge (t = X \vee !bY \vee auxY) \}$

For P_Y: $\{ bY \ \wedge (t = Y \vee !bX \vee auxX) \}$

➔ $\{ bX \ \wedge (t = X \vee !bY \vee auxY) \} \wedge \{ bY \ \wedge (t = Y \vee !bX \vee auxX) \}$

$= \{simplifying\}$

$bX \wedge bY \ \wedge (t = X \wedge t = Y)$

$=$ { t cannot be equal to both X and Y }

false

➔ We reached a contradiction.

➔ **We have mutual exclusion**

Note: auxX and auxY cannot be true since Px and Py are in their critical sections

Limitations of Peterson's algorithm

- **Can synchronize only two tasks**
  - **(extension for more tasks exists but more complex)**
- **Busy-wait**
  - **wastes CPU cycles**
  - **does not work for single core if the scheduling policy is non-preemptive**

Why?

Operating systems kernels provide **semaphores** and **mutexes** instead

Agenda

- Reminder of lecture 2
- Atomicity, Traces and concurrency
- Synchronization
- Peterson’s algorithm
- Synchronization with mutexes

Mutual exclusion using mutexes

- **Special, two-state (or binary) semaphore:** *mutex*
  (see next lecture for details)

- **A mutex *m* must be initialized to 1**
- *Two operations:*
  - ***Lock**(m): < await(m>0); m := m-1 >*
  - ***Unlock**(m): < m := m+1 >*

The **implementation** of *Lock* and *Unlock* **must guarantee their atomicity**

A semaphore is **always non-negative**.

Other names for *Lock* and *Unlock*: *wait / signal, wait / post, P / V*

Synchronizing using a mutex

mutex m := 1;

```
PX = |[
    while (true) {
        do_something();
        lock(m);
        x := x+1;
        unlock(m);
        do_something_else();
    }
]|
```

||

```
PY = |[
    while (true) {
        do_something();
        lock(m);
        x := x+1;
        unlock(m);
        do_something_else();
    }
]|
```

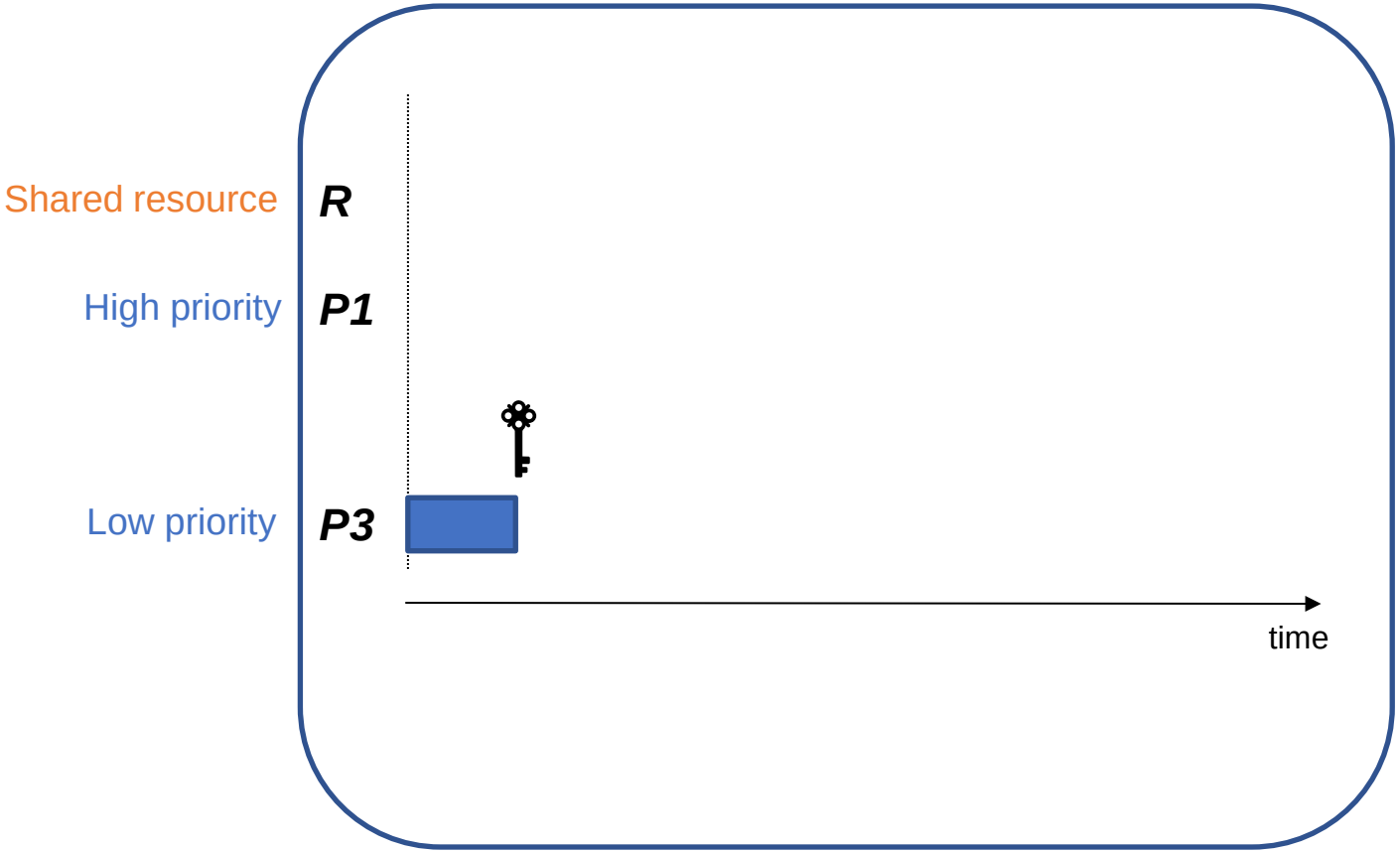
Mutual exclusion using mutexes (POSIX)

```
pthread mutex_t m = PTHREAD_MUTEX_INITIALIZER;
                                    /* static initialization, not always possible */
status = pthread_mutex_init (&m, attr); /* attr: NULL; should return 0     */
status = pthread_mutex_destroy (&m);  /* should return 0                   */
status = pthread_mutex_lock (&m);       /* should return 0                 */
status = pthread_mutex_trylock (&m);  /* returns EBUSY if m is locked  */
status = pthread_mutex_unlock (&m);   /* should return 0                 */
```

Challenges with priority scheduling when protecting shared resource accesses

Blocking

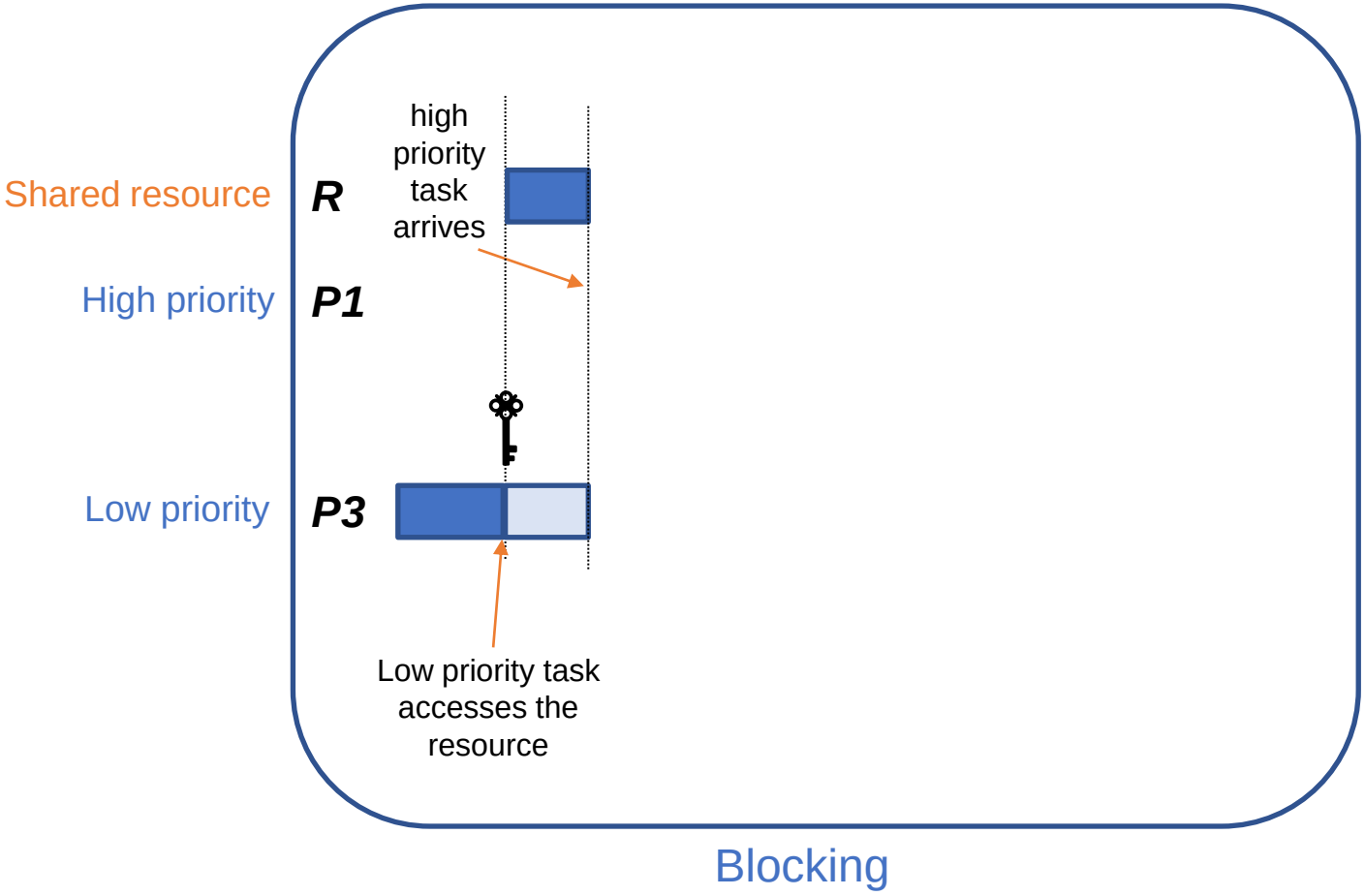
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Blocking

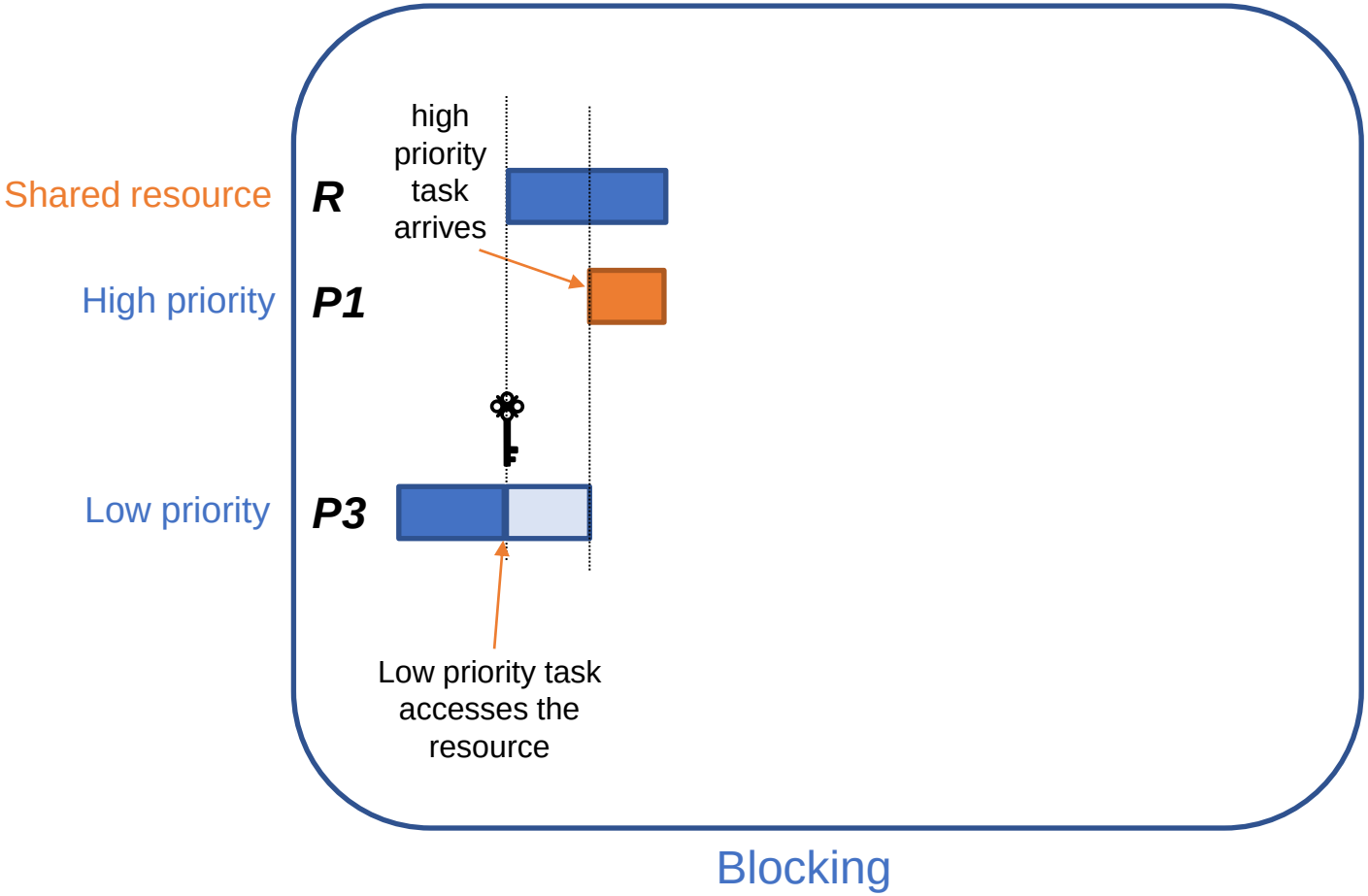
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Blocking

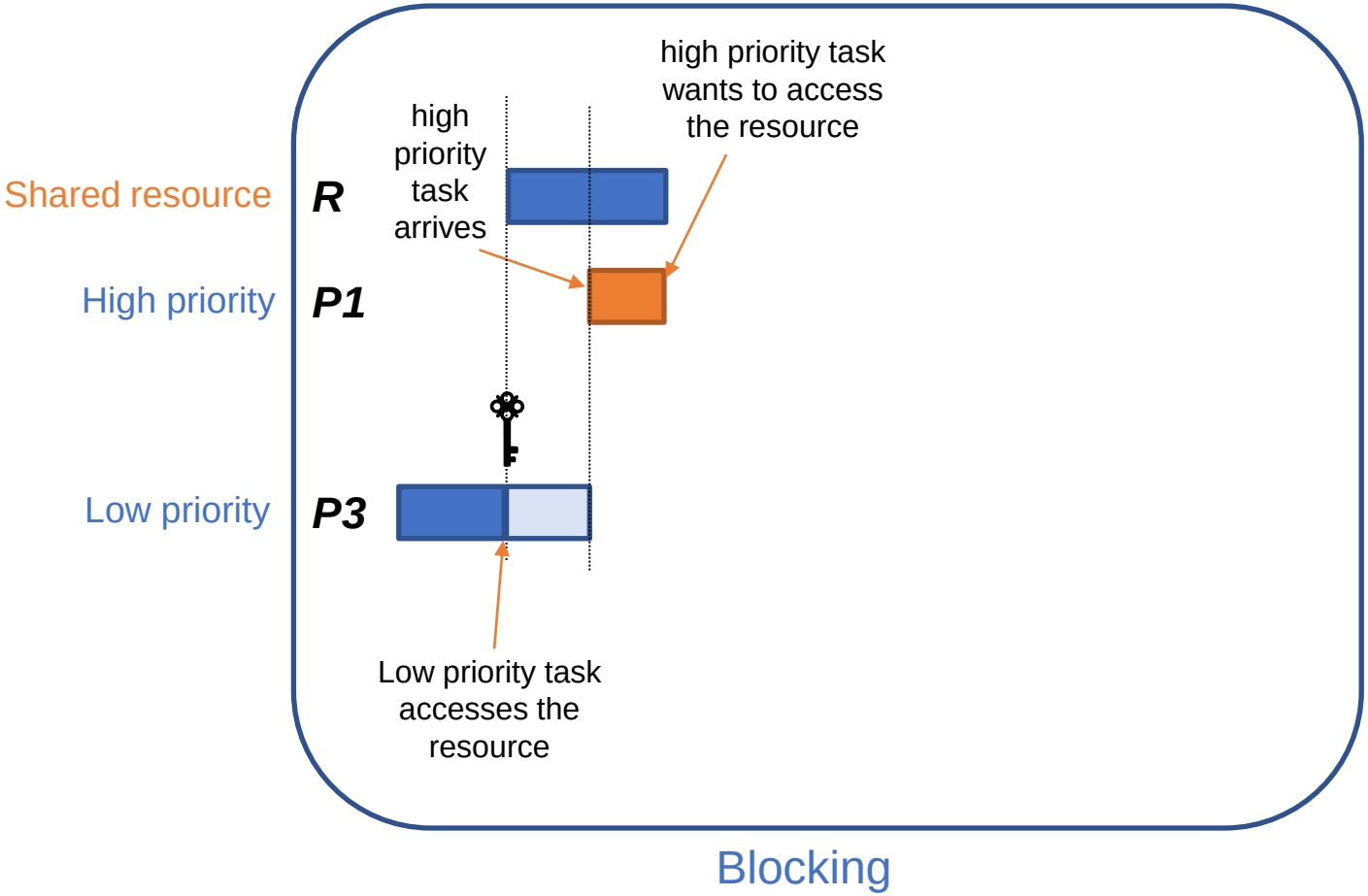
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Blocking

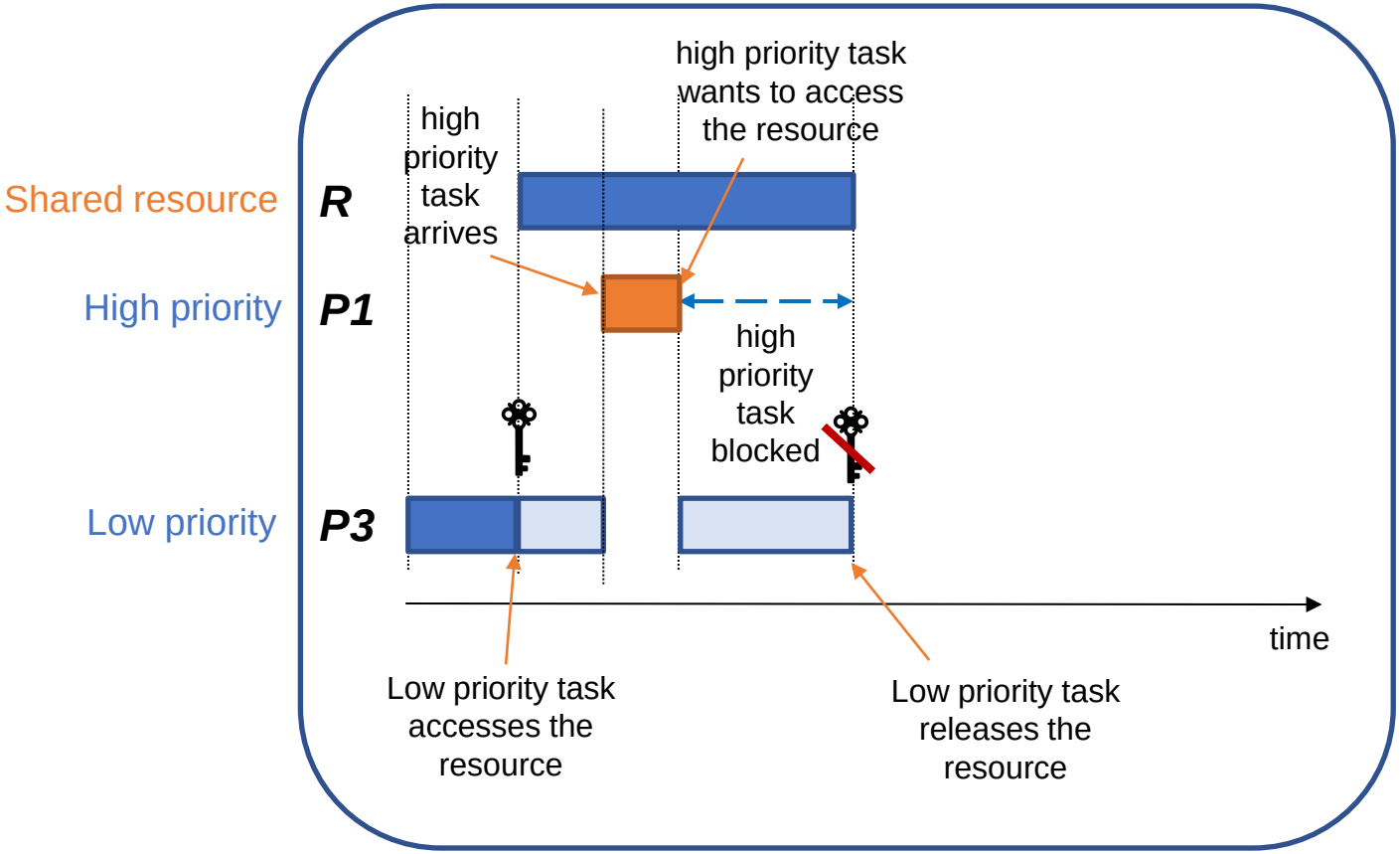
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Blocking

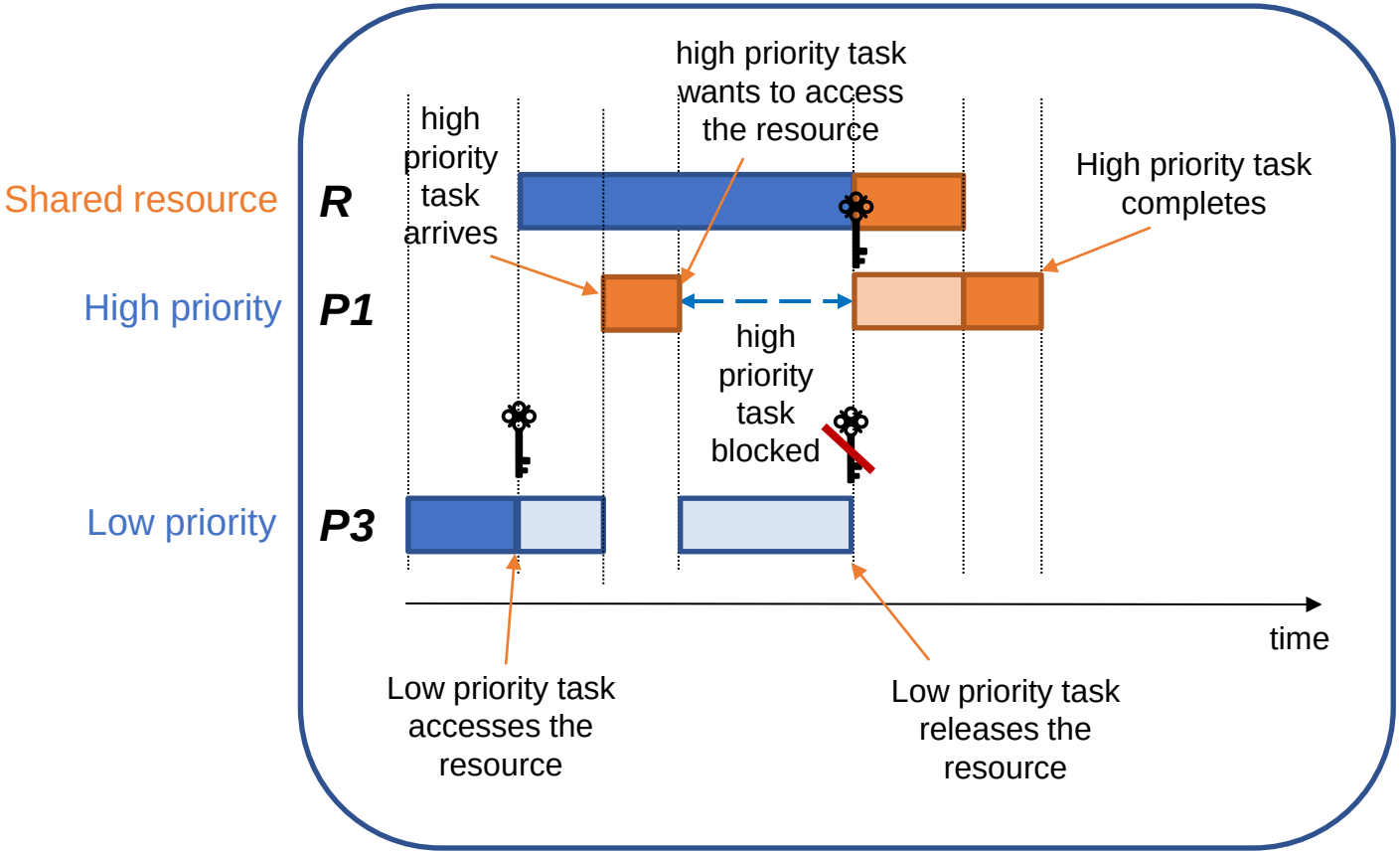
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Blocking

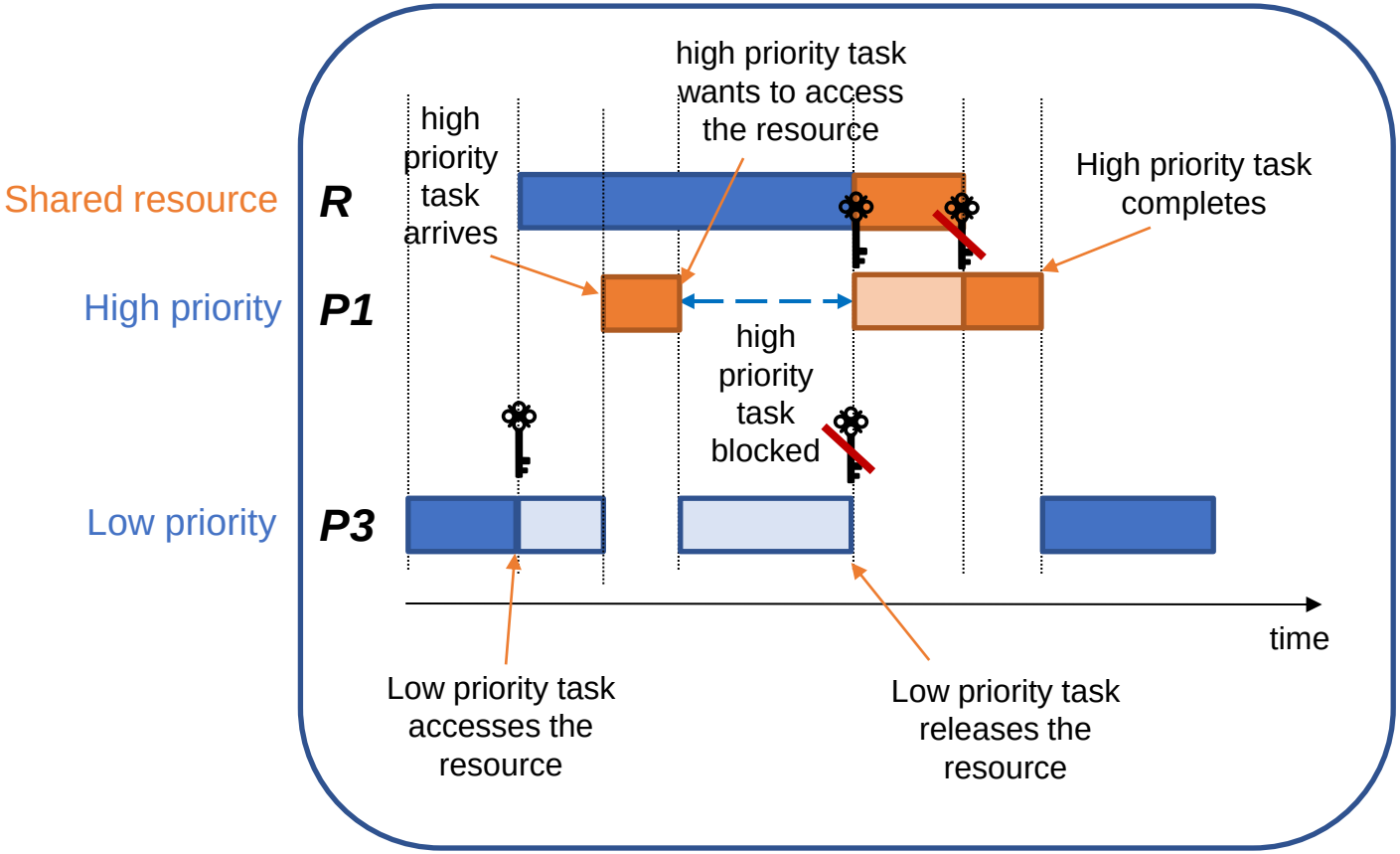
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Blocking

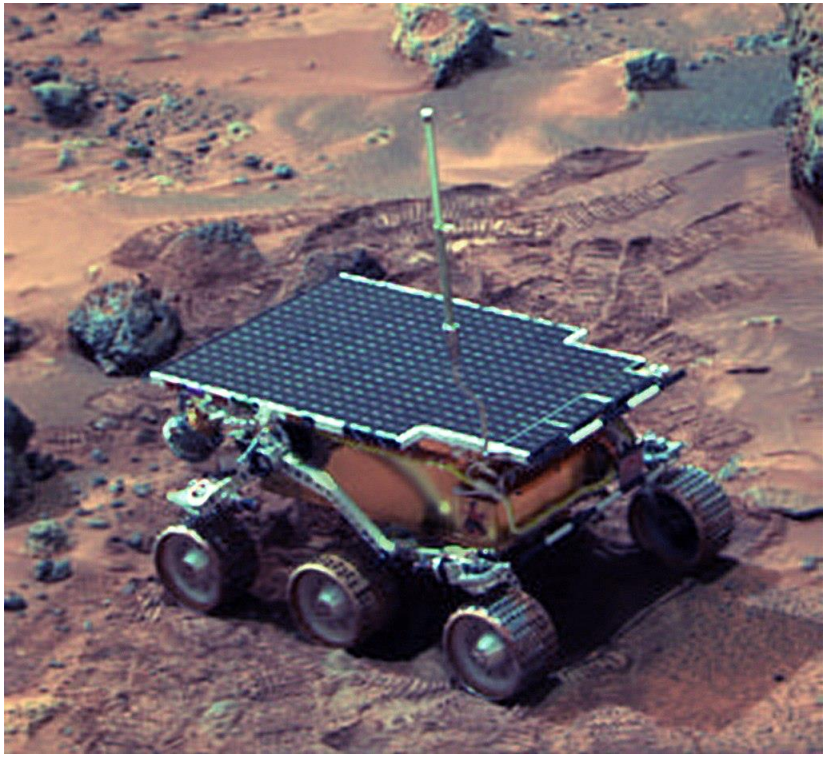
- Blocking:
  - A low priority task obtains a resource; a high priority task then waits on it



Blocking

Priority inversion

- **Mars Pathfinder**

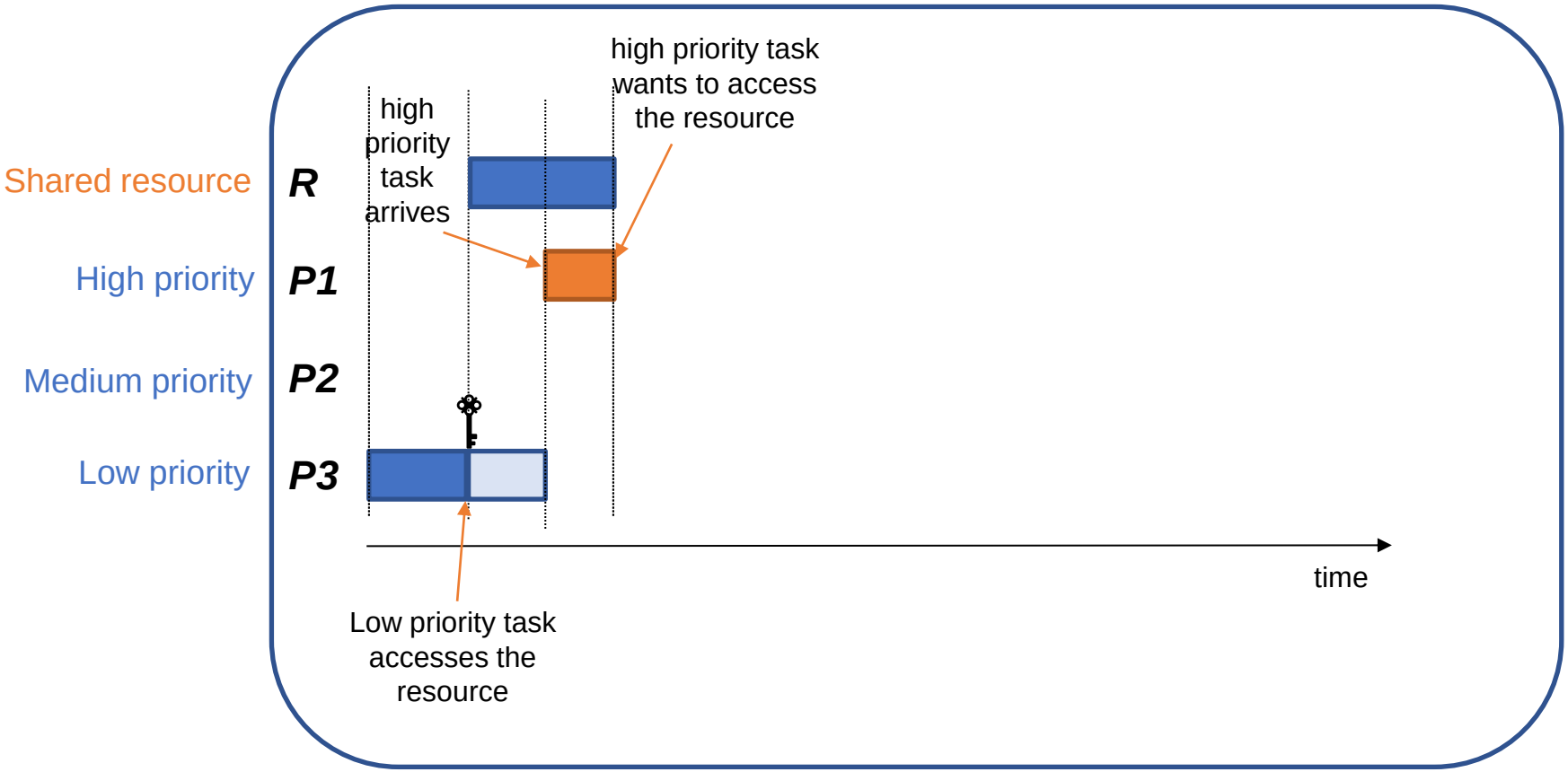


Copyright: NASA - http://photojournal.jpl.nasa.gov/catalog/PIA01122

What really happened on Mars? Authoritative account:
https://web.archive.org/web/20150611035733/http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

Priority inversion

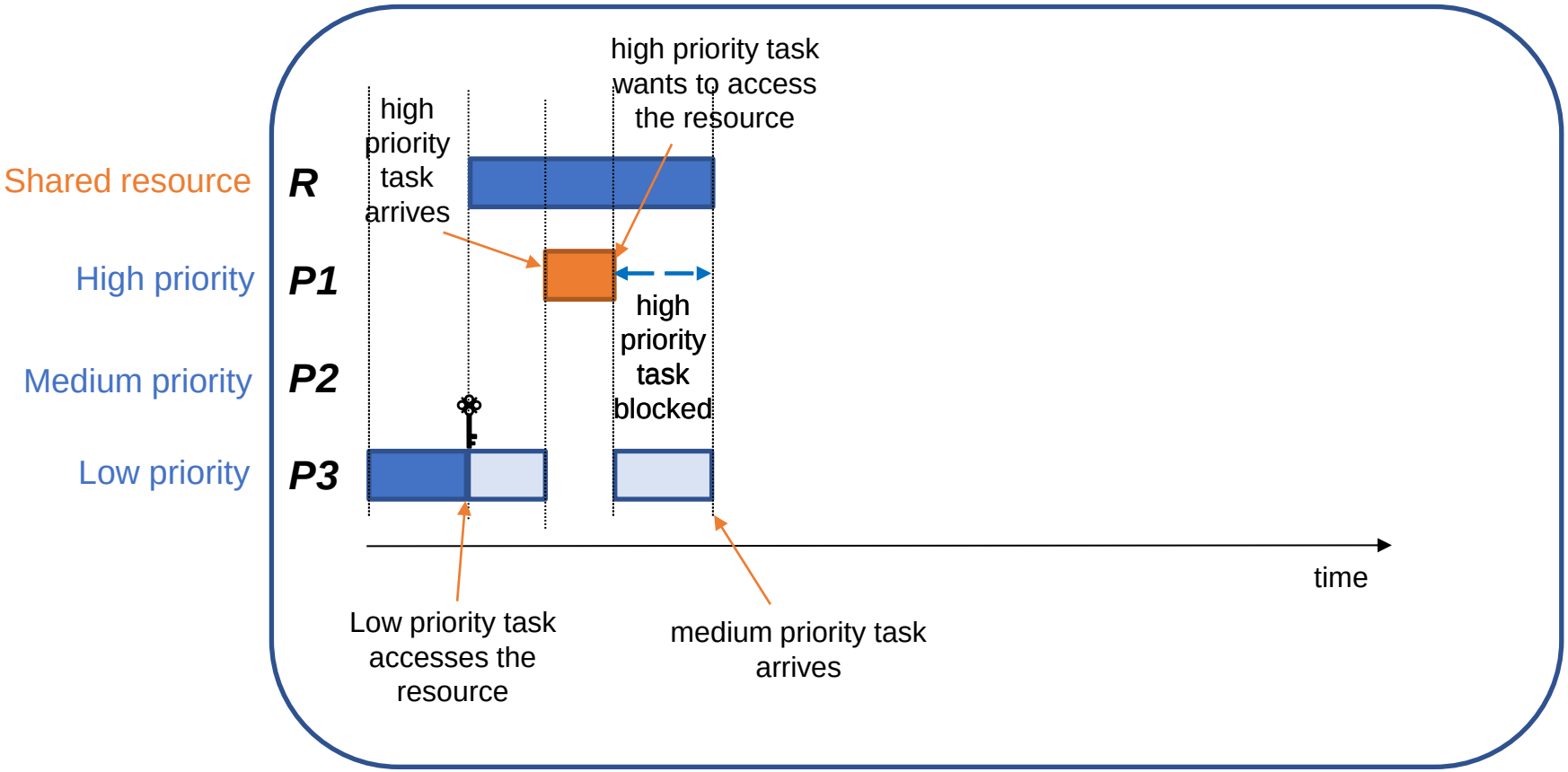
- Priority inversion:



Priority inversion

Priority inversion

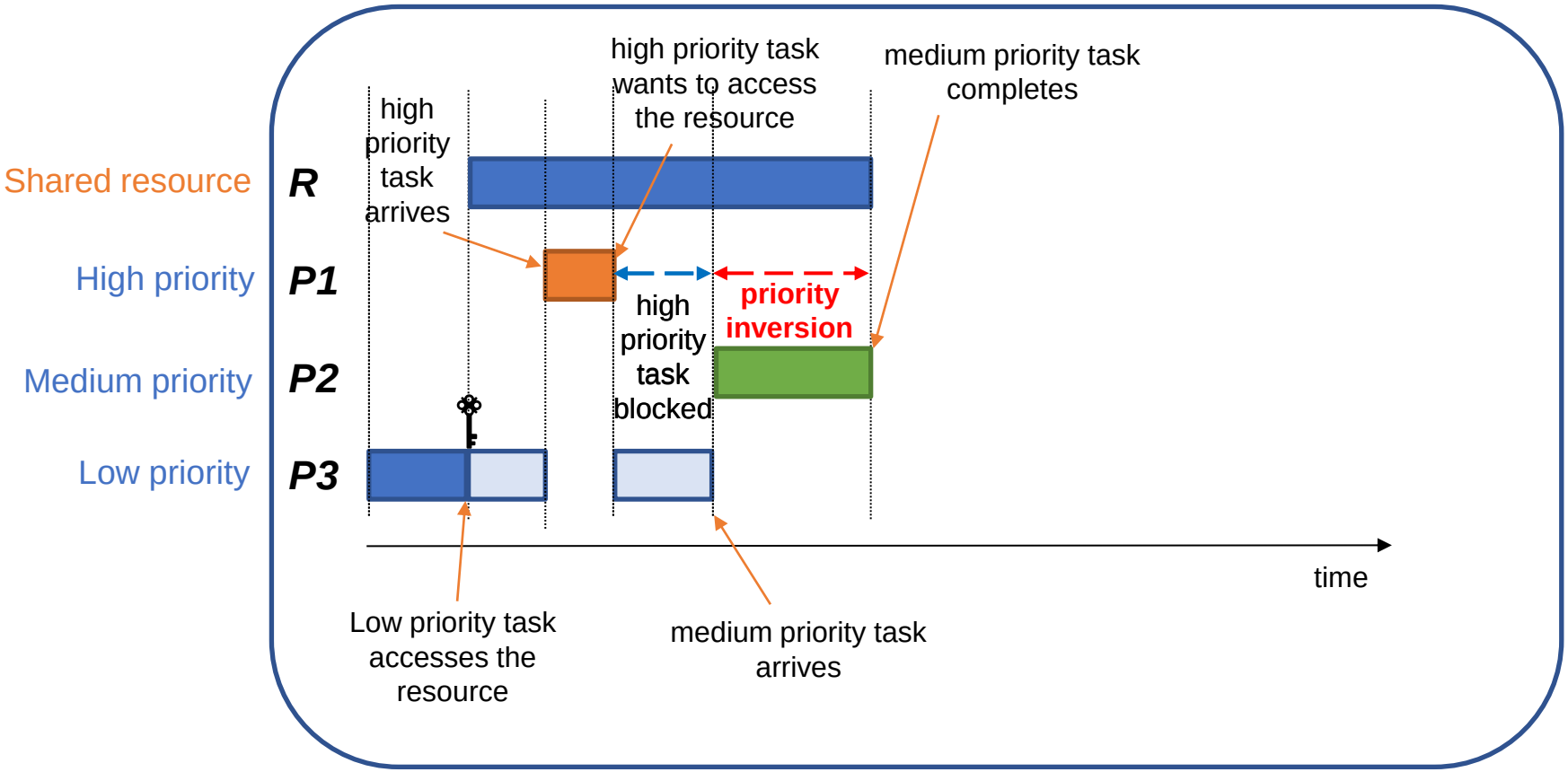
- Priority inversion:



Priority inversion

Priority inversion

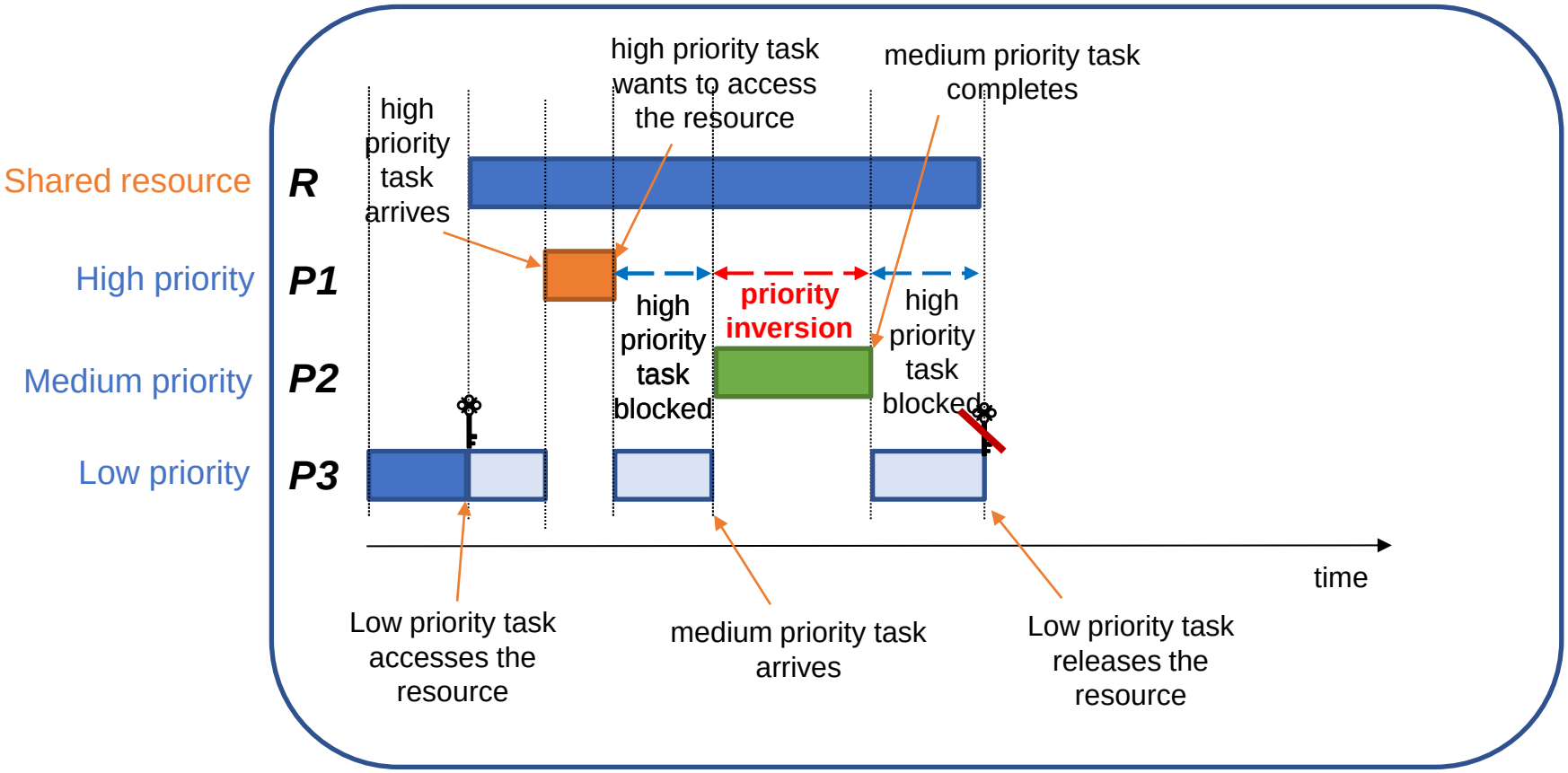
- Priority inversion:



Priority inversion

Priority inversion

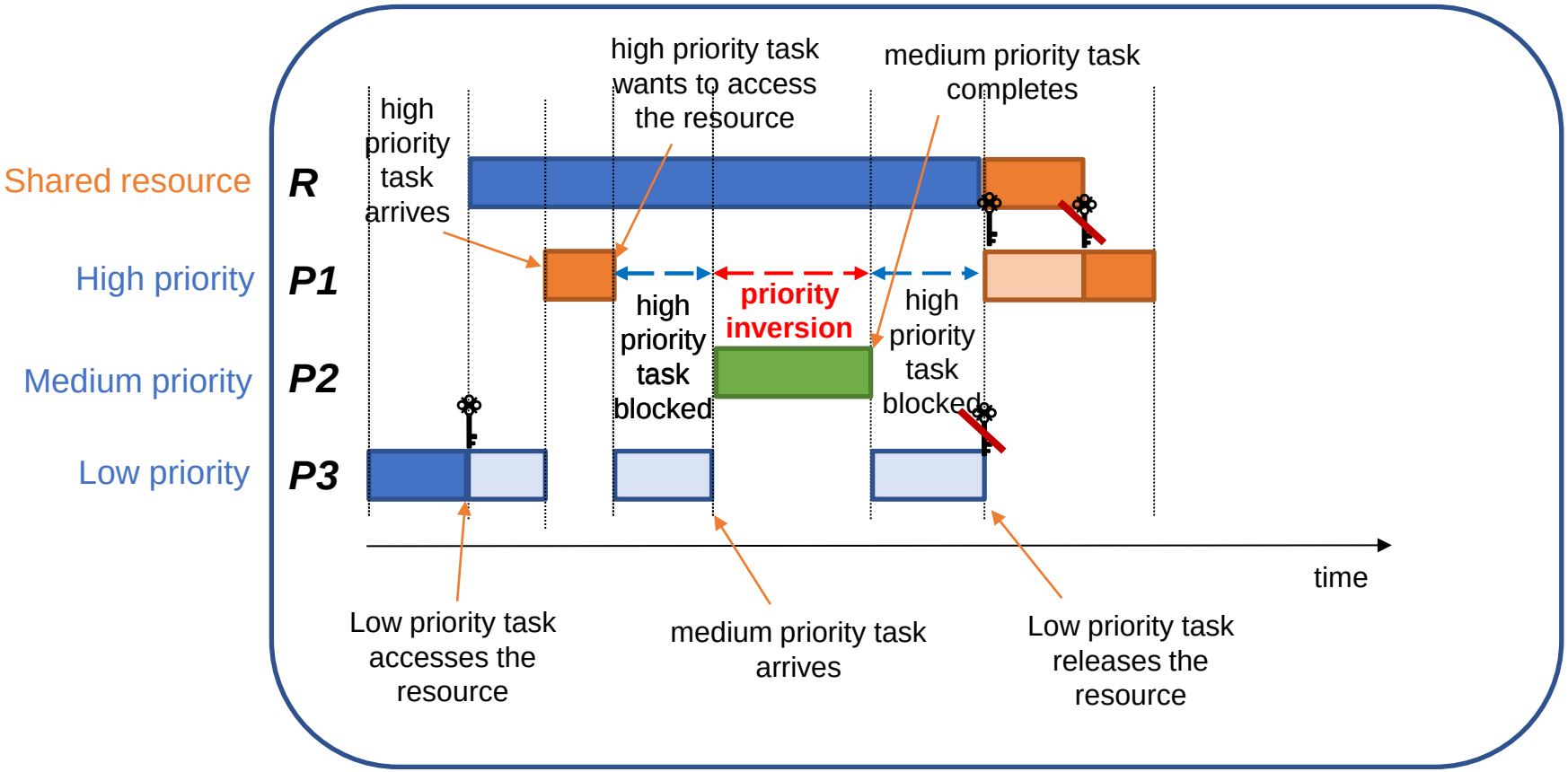
- Priority inversion:



Priority inversion

Priority inversion

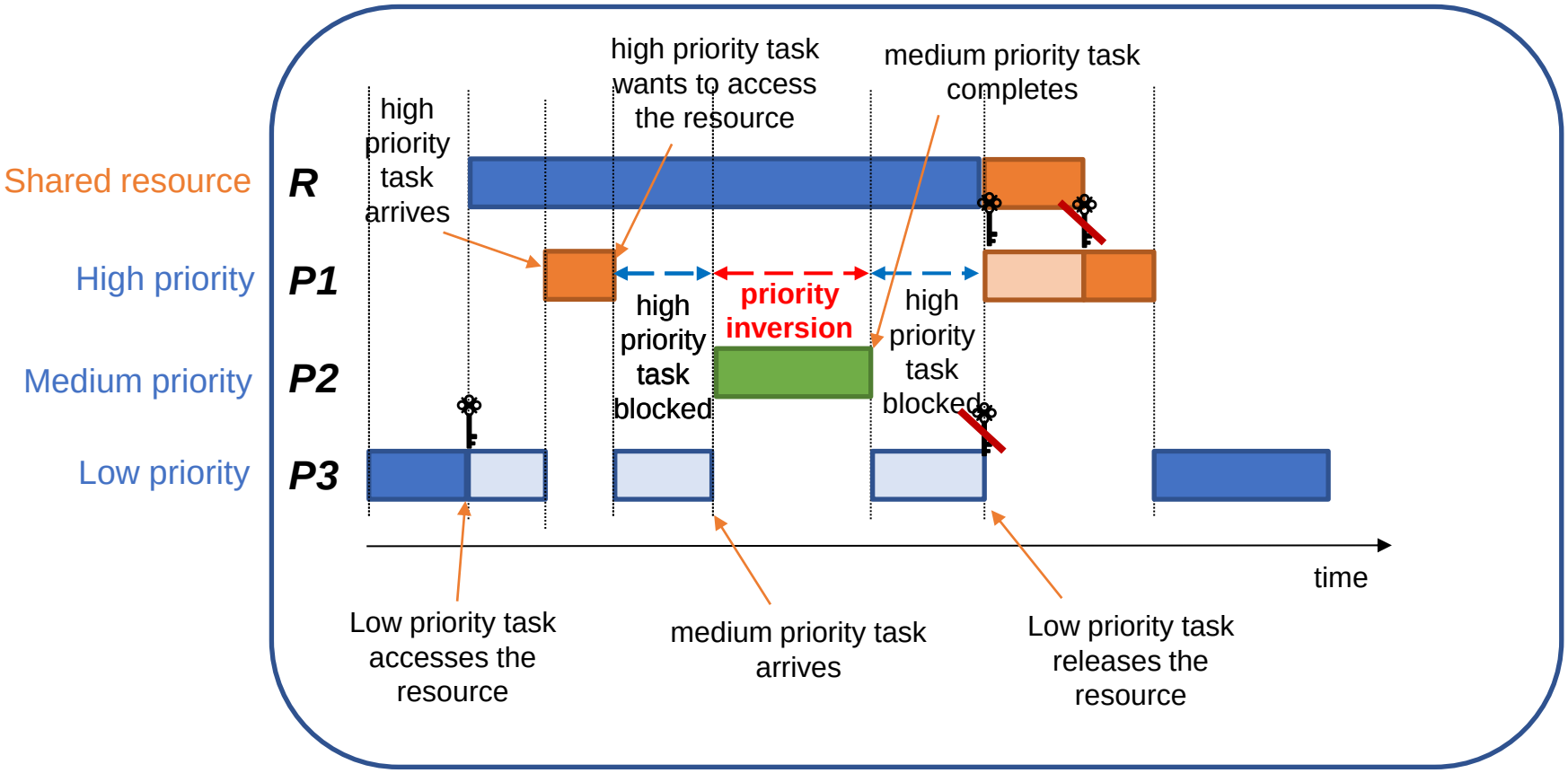
- Priority inversion:



Priority inversion

Priority inversion

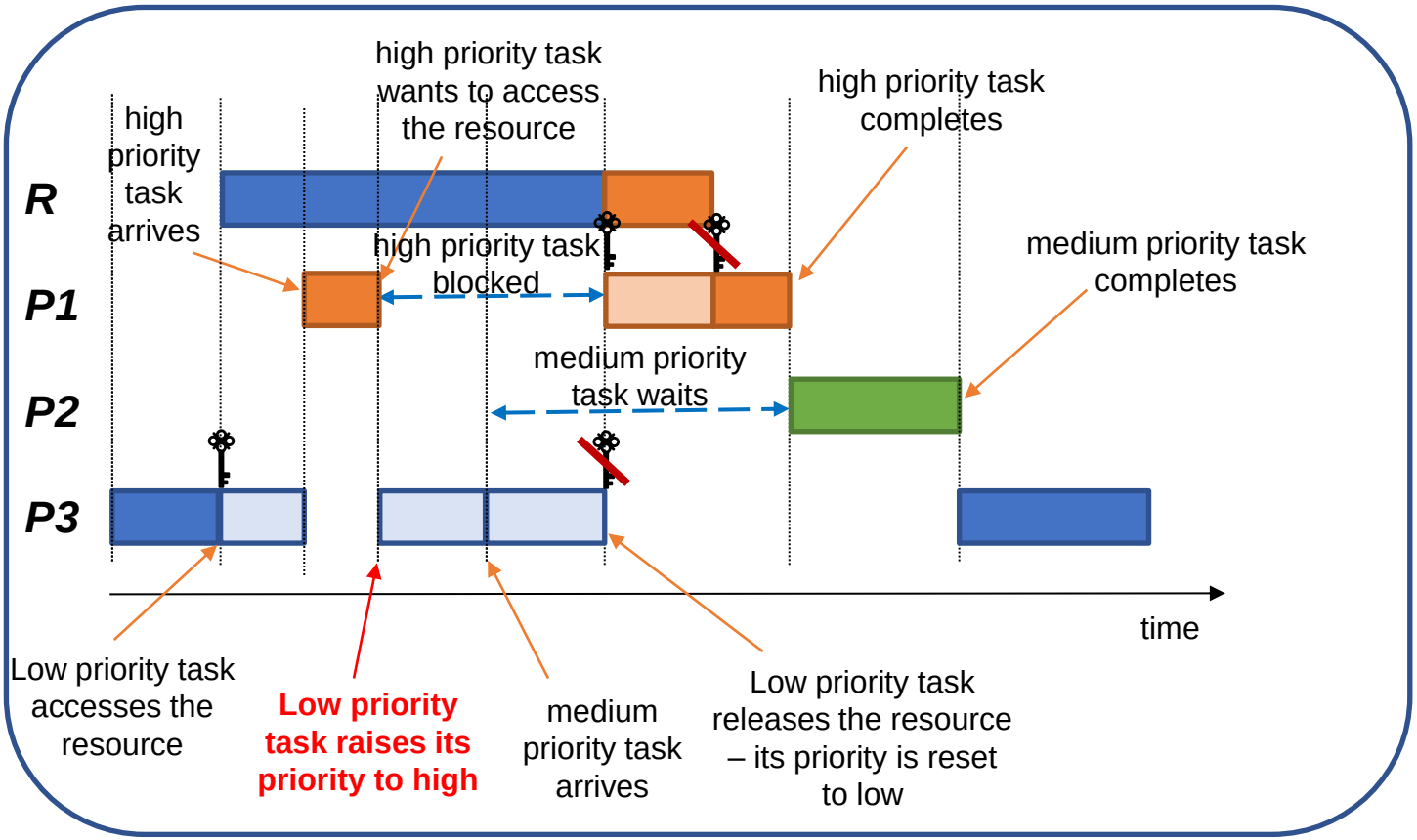
- Priority inversion:
  - A middle priority task pre-empts the low priority task that holds the resource
    - the ***high priority task*** now waits on the middle priority task and executes effectively at the low priority



Priority inversion

A solution to priority inversion: priority inheritance protocol

- The **priority** of the task *P* using the resource is ***dynamically adjusted*** to be the maximum of the priority of any other task that is blocked on the allocated resources of *P* and its own priority
- ...middle priority tasks will wait now.



Priority inheritance

Summary

- Atomic actions and traces
- Issue with concurrency ➔ race conditions
- Synchronization using Peterson's algorithm
- 5 correctness requirements
- Proof for mutual exclusion and absence of deadlock using a proof by contradiction

- Preparation for next lecture (Wednesday)
  - 3 short videos (total: 20 minutes)
  - Topic: actions synchronization using semaphores